# Framework Tasking Notes

## Preface

The following notes are not ready as a releasable document. The main purpose is to provide guidance on how to use the Cloud Computing Data Processing (CCDP) framework for tasking. The organization is based on normal flow of events during a task. The information provided in this paper includes class diagrams, data structures, functionalities, assumptions, and any other information believed to be useful when using the CCDP framework.

## Launching

The framework can be launched either from the command line or by using Mesosphere. The main different between the two launching techniques is that Mesosphere will make sure the framework is running even if stops responding. The class used to launch the framework is com.axios.ccdp.mesos.fmwk.CCDPEngineMain. Running this class with the -h or --help produces the following message:

usage: class com.axios.ccdp.mesos.fmwk.CCDPEngineMain

-c,--config-file <arg> Path to the configuration file. This can also

be set using the System Property

'ccdp.config.file'

-h,--help Shows this message

-j,--jobs <arg> Optional JSON file with the jobs to run

As mentioned in the message this class only requires a configuration file. The configuration file can be passed as command line argument or by setting the Java System Property ‘ccdp.config.file’. The path can contain valid environment variables as they are expanded prior being used. The framework can also launch tasks once it has started if they are provided using the optional --jobs argument.

## Tasking

After instantiation, the CCDP framework waits for incoming tasks from external sources. The framework uses an actual implementation of the CcdpTaskingIntf. This interface allows the framework to register and unregister for channels or topics as well as to send and receive messages or tasks. The default or initial implementation of the interface used by the time of this writing is ActiveMQ.

There are two different types of jobs that can be sent to the framework: Threads and Tasks. A Thread is a list of Tasks that needs to be launched sequentially in the order it is provided. A Task is a single job that can be executed any time and multiple Tasks can run in parallel. In other words, if the json task contains a list of Tasks, then all of the jobs will be executed immediately. On the other hand, if the json task contains a Thread, or a list of Threads, then it will execute the Tasks one at the time in sequence. Because most JSON parsers do not guarantee the order of an array, the Thread needs to know what is the first Task to execute. Appendix A shows examples of the JSON containing a Thread and a list of Tasks. The section below provides a short description of the fields used for tasking.

### Tasking Field Description

* **Threads**: A list of Thread or sequential tasking
* **Thread**: A single sequence of tasks to be executed in order and one rather than in parallel
  + **ThreadId**: A unique identifier for this job for tracking purposes
  + **Name**: A more human readable text to facilitate its identification
  + **StartingTask**: The TaskId of the first task to execute in this sequence. This is needed as the order of the Tasks is not guarantee to be the same as the JSON structure
  + **Description**: A brief description of what this thread does
  + **replyTo**: The channel to use to send status back to the sender
  + **Tasks**: The list of tasks or jobs to execute
* **Task**: A single job or command to run in order to accomplish a specific goal
  + **TaskId**: A unique identifier for this task for tracking purposes. If not provided then the framework generates a UUID and assigns it to the Task.
  + **Name**: A more human readable text to facilitate its identification
  + **ClassName**: The name of the class to use if is an internal module
  + **NodeType**: The type of node to use to run this task (default is ec2). Used to support additional types such as hadoop, elastic-search, EMR, etc.
  + **replyTo**: The channel to send status back to the requester. If not specified, then the framework would use the one specified in the Thread. If a channel is not specified, then the framework does not provide status back to the requester
  + **CPU**: Sets the amount of CPU required to run this task. There are special considerations on this field as explained below.
  + **MEM**: The amount of memory to be reserved in order to be able to execute this task
  + **Configuration**: An optional configuration parameters to provide to the task prior running it
  + **Command**: (**required**) A list of arguments used to construct the command to execute
  + **InputPorts**: A list of ports or way to get data into the task. Mostly used when running a Thread
    - **PortId**: A unique identifier for this port. This is required to map ports from multiple tasks together
    - **FromPort**: The PortId responsible for sending data to this task
  + **OutputPorts**: A list of ports or way to get data out the task. Mostly used when running a Thread
    - **PortId**: A unique identifier for this port. This is required to map ports from multiple tasks together
    - **ToPort**: The PortId expecting the data generated from this task.

The only required field is the Command in the Task. All other fields are optional and therefore can be safely omitted. Most of the fields have default values or are used only for ease of reading a JSON tasking file. The Tasking Allocation section below describes the different scenarios in which a task is executed based on the value of the CPU field.

### Tasking Allocation

As mentioned above, most fields have a default value and therefore do not need to be provided. A special case is the CPU field. This field is used to determine how and where the task is launched based on its value. The following table describes the behavior of the framework based on the CPU resource allocation

|  |  |
| --- | --- |
| CPU Allocation | Effect |
| 0.0 | This is the default value and it executes the task based on a given resource allocation scheme. For instance, the framework decides where this task can be launched using existing resources. |
| 0 > cpu < 100 | If a value is provided and is between 0 and 100 non-inclusive, then the framework will run the task on an existing resource with available CPU resources. |
| 100 | It forces the framework to start a new Virtual Machine and run this task in it. Setting the CPU to 100 assures the user that this task will be the only task launched on the Virtual Machine. |

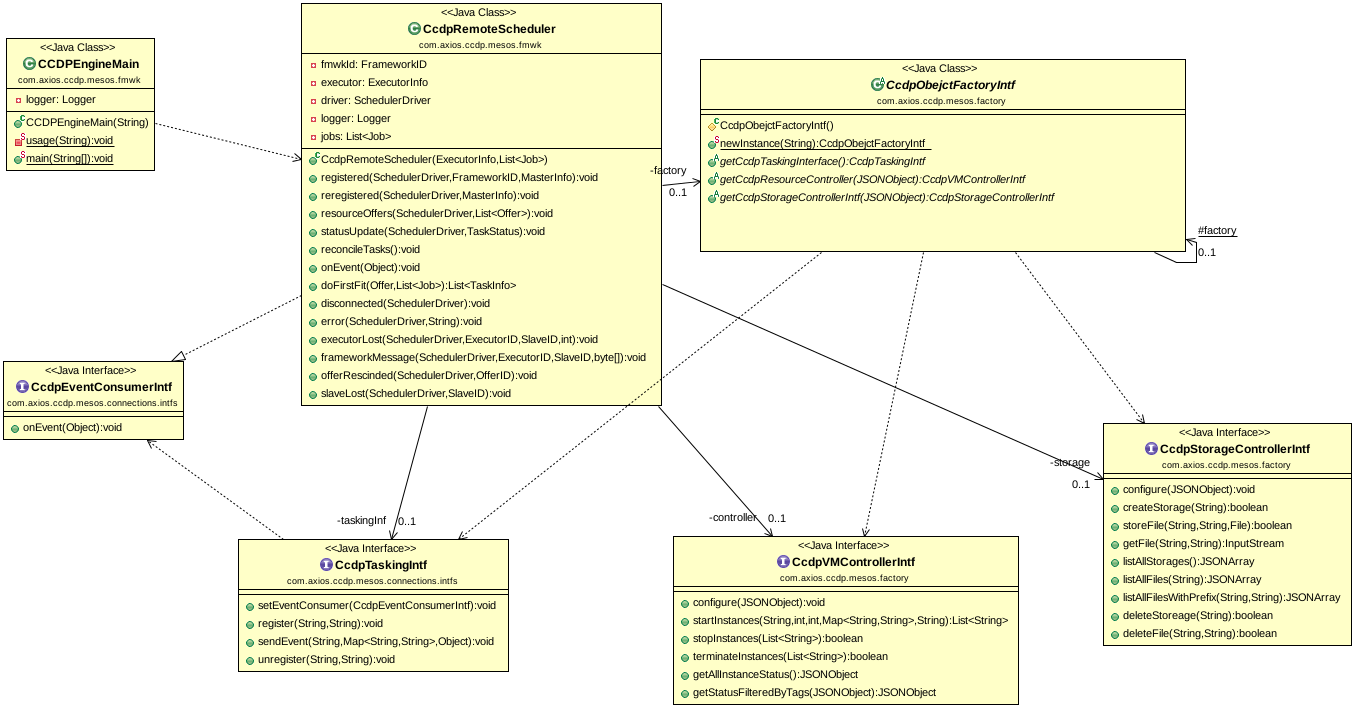
Table 1. Controlling how and where tasks are launched based on the CPU allocation

## Mesos Implementation Data Processing Framework

As it was stated previously, the use of Mesos in combination with Mesosphere Marathon and ZooKeeper allows us to provide a highly scalable, reliable framework. The scalability comes from being able to add EC2 instances as needed dynamically. ZooKeeper runs an application on a Master node and additional nodes can be added in standby mode. This paradigm allows us to add reliability by letting ZooKeeper start Marathon and assure is always running. Marathon is a framework used as the standard way to deploy long running applications including other Mesos Frameworks. Because of that Marathon will be in charge of launching the CCDP Mesos framework.

CCDP provides an EC2 image that can be configured to add itself to the Mesos cluster when is first instantiated. Such capability is what allows CCDP to add more processing nodes as needed responding to an increasing demand of processing power.

The internals of how CCDP process tasks and allocate/de-allocate resources is depicted in figure 1. This class diagram offers a view the most important classes to accomplish its goal only rather than a comprehensive list.

  
Figure 1. Mesos implementation class diagram

Once Marathon is up and running it will receive the task to run an executable jar containing the CCDPEngineMain as the main class. This class sets up some simple initial values and starts the CCDP scheduler (CCDPRemoteScheduler). This class extends the Scheduler interface which is the core interface to interact with Mesos. Implementing such interface is what allows the CCDP framework to receive information of the resources available in the cluster as well as general system status. It also provides additional information such as messages from the agents or agents no longer available.

The CCDPRemoteScheduler uses an actual implementation of the CcdpObjectFactoryAbs abstract class to instantiates the objects needed to perform all the required operations. The CcdpObjectFactoryAbs follows the Factory Method design pattern. Using this pattern decouples the system from actual implementations permitting the use of different Cloud environment without affecting the system itself. An instance of the factory is obtained by calling the newInstance( classname ) method in order to provide a way to implement a Singleton pattern if needed. Singletons are a good approach to ensure that all the entities using the Factory have the same configuration within the same Java session. The CcdpObjectFactoryAbs is an abstract class that provides the newInstance( classname ) method. This method uses Reflection to instantiate a new factory object using the string argument to determine the name of the actual class implementation. This technique makes it easier to swap factories as it is done outside the source code in a configuration file.

All the actual implementations of this factory abstract class need to implement the remaining methods required to create the following objects:

* CcdpTaskingIntf,
* CcdpVMControllerIntf, and
* CcdpStorageControllerIntf

### CcdpTaskingIntf

Simple interface used to receive tasking from external sources or applications. This interface is used to register and unregister from channels or topics where the requests are sent. It also provides the framework with a way to send and receive event or tasking data into and out of the processing framework. The CCDP Scheduler implements the CcdpEventConsumerIntf interface. This interface is the only requirement this class has in order to instantiate a CcdpTaskingIntf object. Each CcdpTaskingIntf implementation knows how to connect and receive the tasks. Every time a task arrives it invokes the onEvent() to notify the scheduler. Example of actual implementations of this interface would be JMS, File System, Sockets, etc.

### CcdpVMControllerIntf

While the CcdpRemoteScheduler concentrates on keeping track of all the tasks and task allocation, an instance of this class focus on starting and stopping virtual machines. Implementing this class requires knowledge of the business logic to determine when to start or stop a virtual machine dynamically. Removing this responsibility from the scheduler simplifies substituting this type of logic to accommodate for different scenarios or to try different approaches in order to determine the best solution. Interchanging implementations should not have any impact on the scheduler implementation. A simple example of implementing this class could be based on resources’ watermark. For instance, if the combined CPU and/or memory are above 65% for more than one minute then launch a new VM and if the number of task in the last 5 minutes is zero then shut down the virtual machine.

It is important to remember the special cases provided by setting the CPU to 0, 100, or a value in between as described earlier can also be accommodated here.

### CcdpStorageControllerIntf

There are occasions where a task needs to write/read results generated on a different virtual machine. Using this interface separates the how from the scheduler adding a more flexible approach to different storage solutions. For instance, a file generated by a task can be written to an NFS mount disk or stored on a S3 bucket to be accessed by another task.

## Tasking Execution

Once a Virtual Machine has been identified, the task is executed by the CCDPCommandExecutor class. This class extends the Executor class required by Mesos to communicate with the system and to receive tasks. This class is also responsible for sending status back to the CcdpRemoteScheduler indicating the progress of the tasking execution. When a status change during the task execution, the executor sends one of the following task states:

* TASK\_DROPPED
* TASK\_ERROR
* TASK\_FAILED
* TASK\_FINISHED
* TASK\_GONE
* TASK\_GONE\_BY\_OPERATOR
* TASK\_KILLED
* TASK\_KILLING
* TASK\_LOST
* TASK\_RUNNING
* TASK\_STAGING
* TASK\_STARTING
* TASK\_UNKNOWN
* TASK\_UNREACHABLE

This information is sent back to the requester if a replyTo field is provided.

## Appendix A: JSON Tasking Examples

**Listing 1**. Example of a list of Threads. Executes the Tasks one at the time

{"Threads":

[

{"ThreadId": "thread-1",

"Name": "PI Estmator",

"StartingTask": "cycles\_selector",

"ReplyTo": "The Sender",

"Description": "Estimates the value of PI",

"Tasks":

[

{"TaskId": "cycles\_selector",

"Name": "Cycles Selector",

"ClassName": "tasks.emr\_demo.CyclesSelector",

"NodeType": "ec2",

"ReplyTo": "The Sender",

"Command": ["python", "/opt/modules/CsvReader.python", "${CCDP\_HOME}/data/csv\_test\_file.csv"],

"Configuration": { "number-cycles": "10000", "wait-time": "5" },

"InputPorts": [],

"OutputPorts": [{"PortId": "cycles\_selector-1", "ToPort": [ "pi\_estimator\_input-1" ] } ]

},

{"TaskId": "pi\_estimator",

"Name": "PI Estimator",

"ClassName": "tasks.emr\_demo.PiEstimator",

"NodeType": "emr",

"ReplyTo": "Someone Else",

"Command": ["python", "/opt/modules/tasks/csv\_demo/JsonTranslator.py", "-i", "${CCDP\_HOME}/data/input\_text.txt", "-o", "${CCDP\_HOME}/data/out\_text.json"],

"Configuration": { "output-file": "${CCDP\_HOME}/data/pi\_values.csv" },

"InputPorts": [{"PortId": "pi\_estimator\_input-1", "FromPort": [ "cycles\_seletor\_output-1" ] } ],

"OutputPorts": [{"PortId": "pi\_estimator\_output-1", "ToPort": [ "pi\_reader\_input-1" ] }]

},

{"TaskId": "pi\_reader",

"Name": "PI Value Reader",

"ClassName": "tasks.emr\_demo.PiReader",

"NodeType": "ec2",

"ReplyTo": "The Sender",

"Command": ["python", "/opt/modules/tasks/csv\_demo/JsonReader.py", "-i", "${CCDP\_HOME}/data/out\_text.json"],

"Configuration": {"input-file": "${CCDP\_HOME}/data/pi\_values.json", "wait-time": "5" },

"InputPorts": [ { "PortId": "pi\_reader\_input-1", "FromPort": [ "pi\_estimator\_output-1" ] } ],

"OutputPorts": []

}

]

}

]

}

**Listing 2**. Example of a single Task

{

"Tasks":

[

{"TaskId": "csv\_reader",

"Name": "Csv File Reader",

"ClassName": "tasks.csv\_demo.CsvReader",

"NodeType": "ec2",

"ReplyTo": "The Sender",

"CPU": "10",

"MEM": "128",

"Command": ["python", "/opt/modules/CsvReader.python"],

"Configuration": { "filename": "${CCDP\_HOME}/data/csv\_test\_file.csv" }

}

]

}

**Listing 3**. Example of a list of Tasks, these tasks will be executed at the same time based on resources availability

{"Tasks":

[

{

"CPU": "10",

"MEM": "128",

"Command": ["python", "/opt/modules/CsvReader.python", "${CCDP\_HOME}/data/csv\_test\_file.csv"]

},

{

"CPU": "100",

"MEM": "128",

"Command": ["python", "/opt/modules/tasks/csv\_demo/JsonTranslator.py", "-i", "${CCDP\_HOME}/data/input\_text.txt", "-o", "${CCDP\_HOME}/data/out\_text.json"]

},

{

"Command": ["python", "/opt/modules/tasks/csv\_demo/JsonReader.py", "-i", "${CCDP\_HOME}/data/out\_text.json"]

}

]

}